Telecommunications
Internship Report 2A

GreenSDN

SDN Low-Cost Test-bed For Saving Energy Application in Data Center Networks with ElasticTree Use Case

Student :
Julien Mathet
jmathet@enseirb-matmeca.fr

Supervisor :
Adel Nadjaran Toosi
Adel.N.Toosi@monash.edu

17 June - 6 September 2019

Contents

List of Figures

Acknowledgements

For my internship, I want to express my gratitude to my internship supervisor who has been a huge help throughout my work. Adel N. Toosi spent a lot of time with me, which allowed me to progress at a very good pace.

For the future of this project, I want to thank Tuhin Chakraborty, a PhD student who is going to use my work as the basis for his thesis. He really helped me to improve my report.

Introduction

The emergence of social networks, video streaming, or IoT devices demands huge network infrastructures to support the increasing amount of traffic. To be able to meet the demand, new and large data centers are built. They are mighty powerful and consume a large amount of energy, which, in the current environmental context, is a major source of concern. It becomes more relevant when network devices do not have an energy consumption proportional to their usage. On top of that, the Software Defined Networks (SDN) approach emerges as an agile and flexible solution for computer networks thanks to a centralized view of the network in a single point of management.

This area is the subject of numerous research work, like in Monash University. This is the biggest public Australian University and one the top university in the country. In 2016, 50 000 undergraduate students and 20 000 graduate students were welcomed. It is the most popular university in the state of Victoria. In addition, Monash hosts more than 100 research centers divided into 10 faculties. The staff produces over 3 000 research publications each year in over 150 fields of different studies.

The Faculty of Information Technology (IT) is one of them, it is dedicated to research in a wide range of topics such as networks, cloud computing, machine learning for example. Dr Adel N. Toosi is a Lecturer at IT Faculty whose research is guided by software-defined networking, cloud computing and energy efficiency. It is thanks to him that I was able to do a 3-month internship in these fields.

As presented, SDN networks features facilitates the implementation of new applications. But to design, to develop and specially to test solutions. Two methods are mainly used to run experiments: the emulation and the real test-bed. For their costs and their complexities of configuration, the emulation is more popular, compared to real test-beds which enable more realistic experiments. Therefore, parts of my project are focusing on the build of an emulated network and a real and low-cost test-bed for SDN data center networks. Another part is dedicated to the implementation of an energy saving application. The solution developed is named ElasticTree, proposed by Hell et al. [1] in 2010. The idea is to save energy in a data center by switching off a part of the network without compromising the quality of service.

In this way, the implementation of the ElasticTree application will be presented after a review of the state of the art. Then, the emulated network and the results obtained will be detailed. Finally, the low cost test-bed, used to measure the energy efficiency of the application in real environment, will be explained in details.

1 State of the art

1.1 Software Defined Networks

The SDN approach consists of separating the control plane from the data plane (forwarding) through the use of a controller. This is the brain of the network with a centralized view. In order to facilitate network management, it enables a high level of abstraction over the application layer which communicates to the controller via Northbound APIs. These high-level policies are injected to the network devices through low-level rules. To achieve this goal the Open Networking Foundation defines the OpenFlow protocol to formalize the communication between the controller and network devices. However, there is no standard for SDN controller. Many exist nowadays, the most common are Open Daylight, Floodlight, and onOS (used in this project).

1.2 Network Virtualization

The virtualization is the ability to emulate a hardware component using the software. In networks context, the virtualization enables us to be independent of the hardware configuration. Hence, the use of virtualized resources allows more agility and efficiency than a hardware-based solution. The network virtualization abstracts network connectivity with a logical module running independently on top of physical network. This technique reduces the cost of physical resources with the use of generic hardware rather than expensive proprietary network devices. In addition, network virtualization brings plenty of flexibility in data center networks. Resources can be available on demand without hardware modification because a single machine can support multiple virtual devices.

1.3 Mininet, a Network Emulator

one use of the network virtualization is the emulation of all network devices in the same machine. For this purpose, network emulators are used; they are software which makes possible the creation of a virtual network in a single machine. They emulate the behavior of network devices in order to be able to perform tests and experiments without having to support the cost of the network equipment.

In the context of SDN, Mininet is the most popular network emulator software. It enables to lunch a network with hosts, switches, and SDN controller. These virtual switches are programmable using the OpenFlow protocol, which is useful to test SDN applications. In addition to that, we can also validate the results from the emulator by running a small prototype application on real testbeds.

1.4 Real Testbeds

Real testbeds can be used to validate the proper functioning of systems in a real environment. However the cost can quickly increase. That is why the use of network virtualization tools

and cost-effectiveness hardware is often chosen. The article [2] is an example of low-cost testbed, and a micro SDN data center was built using Raspberry-Pis as SDN switches. In order to create a switch in this small computer, the open-source software Open vSwitch (OVS) was used. It enables to implement multilayer virtual switches in every device.

1.5 Data Center Networks

Data centers are the sources of plenty of resources in term of computation, storage, or network. These resources need a scalable and efficient internal network to interconnect a large number of servers. Besides, cloud computing applications generate massive host-to-host traffic. Therefore, data center networks must offer scalability and flexibility while providing high bandwidth and failure tolerance.

In a classic tree topology, one possibility could be to deploy high-rate switches with a significant amount of ports. But it would drastically increase the data center cost. To reduce this cost without deteriorating the performances, new architectures were designed with many equal-cost paths between any given pair of the hosts and small switches (in term of port number). The fat-tree topology [3] belongs to this group. The figure 1.1 represents a $k = 4$ fat-tree topology, k representing the number of ports in each ethernet switch i.e. the degree. In a fat-tree topology, three layers can be defined: core at the top, aggregation in the middle and edge at the bottom. All core layer switches are connected to every pod (k pods in total) containing two layers of $k/2$ switches. Then, every aggregation layer switch is connected to all edge layer switches of the pod. Finally, edge layer switches (lower layer) are connected to $k/2$ hosts.

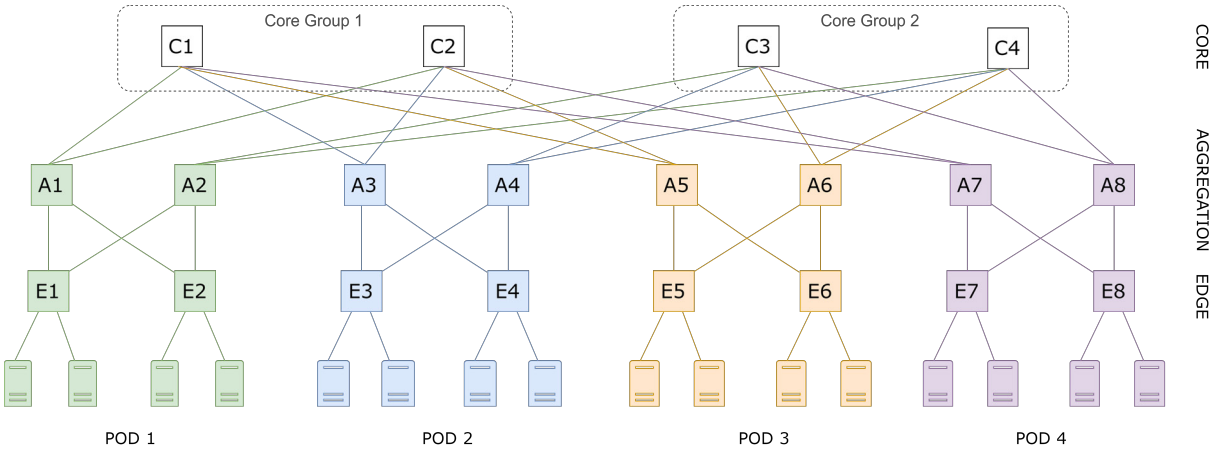

Figure 1.1: $k = 4$ Fat-Tree Topology

1.6 Data Center Workload

As seen before, data center networks are built to provide high performances to satisfy the demand in all circumstances. For this purpose, networks are provisioned for peak workload. Rare events make it possible to reach the maximum load, but most of the time, the data center could operate on a subset of devices. In addition, power management studies [4] had shown that the energy consumption of data center devices increases very slightly with the rise of the workload. This is why turning off a switch is more beneficial than to limit the use (for example turning off some ports). Based on this idea, a solution named ElasticTree was proposed.

1.7 ElasticTree Concept

1.7.1 General Presentation

Due to a large number of redundancies in data center topologies, most of the time links are not fully used. A better traffic monitoring could help to optimize the number of used links. This number can be reduced by concatenating the traffic through selected links. If the amount of unused links is large enough, then some switches can be turned off. The concept of ElasticTree starts with this observation.

The main idea is to monitor data center traffic in order to optimize the set of network elements that must stay active to meet performance and fault tolerance goals [1]. This approach consists of finding the minimal subset of network elements which will be sufficient to forward information across the whole network so that the inactive network elements are switched off to save energy.

The figure 1.2 represents the theoretical system architecture of the ElasticTree model with three logical modules namely optimizer, forwarding and power control. The modules are the following:

1. Traffic data is collected by the **optimizer** in order to find the most energy-efficient subset of devices while taking into account power models for each switch and the desired fault tolerance properties into account;
2. The new subset of network component is transmitted to the **forwarding** algorithm to generate corresponding flow routes;
3. Finally, the new subset is used by the **power control** to switch on needed switches and to switch off unneeded switches.

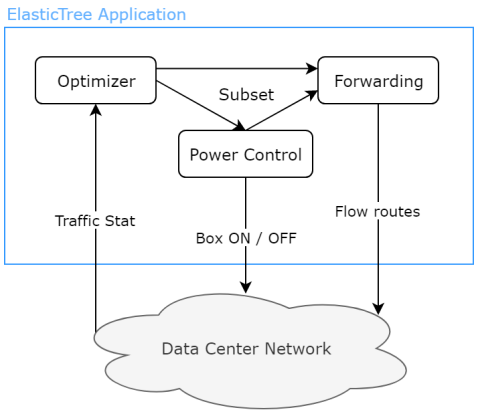

Figure 1.2: ElasticTree logic

1.7.2 Optimizer

This part will focus on various types of optimizer to understand the module challenges. Its purpose is to compute the minimum-power network subset. Three approaches, namely, Formal Model, Greedy Been-Packing, and Topology-aware Heuristic are presented. All of them achieve different trade-offs between scalability and optimality.

First, the **Formal Model** is the most optimal-power solution to find the optimal flow assignment that satisfies the traffic constraints [1]. These constraints are the link capacity, the flow conservation, and the demand satisfaction.

It uses topology, switch power model and traffic matrix [1] as inputs. And, this model outputs the new topology (a subset of the original) and routes taken by each flow to satisfy the traffic matrix. But this computation is not flexible, and the complexity is about $O(n^{3.5})$ where n is the number of hosts.

Secondly, the **Greedy Bin-Packing Method** was designed to improve the scalability of the previous one. "The optimality of the solution is not guaranteed but in practice, very good results are obtained." [1] The principle is to compute path for each flow and to choose the leftmost one (within each layer paths are chosen in a deterministic left-to-right order) with sufficient capacity. The traffic matrix is needed as input and this method returns the active subset (set of devices and links traversed by one or more flows) and each flow path. The scalability was improved in comparison to Formal Method with a complexity between $O(n^{2.5})$ and $O(n^2)$ (respectively for no-split flow version and with-split flow version) [1].

Finally, the **Topology-aware Heuristic** is a method taking advantage of the fat-tree topology regularity. In contrast with the previous methods, it does not compute the set of flow routes and assumes divisible flows. These assumptions allow the algorithm to compute with less information and in a shorter period of time. In addition, the separation of the optimizer and the routing module allows the choice of any fat-tree routing algorithm. This last method finds the subset with only port counters and not the full traffic matrix. The concept here is the following: “to satisfy traffic demands, an edge switch does not care which aggregation switches are active, but, instead, how many are active” [1]. Based on traffic, the number of links required to support up and down traffic can be computed. Then, for the aggregation and core layer, the number of switches needed is directly deducted from the link calculations.

[1] a matrix giving the traffic volumes between origin and destination in a network

2 ElasticTree Implementation

In the context of an SDN network, the programmability of the network is facilitated with the use of a controller. In this project, a plug-in-play application was developed to implement ElasticTree. It contains three modules: optimizer, forwarding, and power control. These modules work together and communicate with the controller through HTTP request to the ONOS REST API. The figure 2.1 represents the architecture of the entire system implemented. It can be noted that the application was developed in Python. In this part, the implementation of the three modules will be detailed.

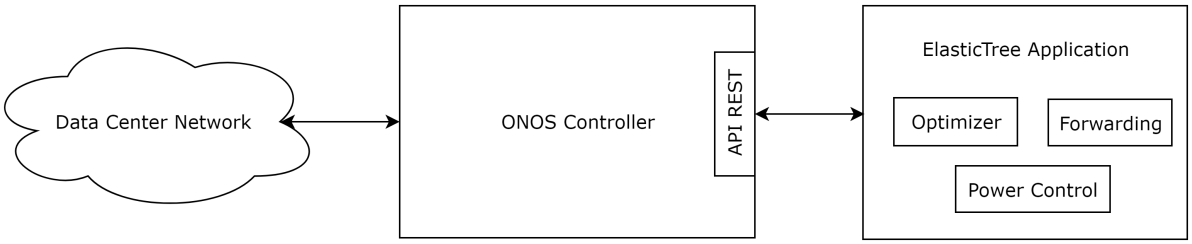

Figure 2.1: General Architecture

2.1 Optimizer

As seen before the aim of the optimizer module is to compute the minimum-power network subset. For this purpose, the **Topology-Aware Heuristic optimizer** presented before was chosen for its scalability and its ability to separate the optimization and the forwarding parts. This optimizer takes advantage of uniform fat-tree topologies by performing computations based on layers. The idea is to compute the number of devices (switches) needed in the aggregation layer of each pod and also, in each group of the core layer. Here, edge layer switches are all necessary to keep the hosts connected.

In order to achieve this, up-traffic and down-traffic rates on each link are used for computations. The method is based on the published literature [1]. We add the concept of core groups to make the optimizer compatible with the forwarding module (presented later).

First, to know the minimum number of active aggregation switches $NAgg_p$ needed in the pod p, the maximum between the number of aggregation switches needed to support the down-traffic and the up-traffic:

$$NAgg_p = \max\left(NAgg_p^{up}, NAgg_p^{down}, 1\right) \tag{2.1}$$

The number 1 is required to guarantee the minimum spanning tree even if there is no traffic. The following method is used to compute the two needed values:

1. $NAgg_p^{up}$ is the number of aggregation switches required to satisfy the up-traffic in the pod p:
   (a) Computation of $LEdge_{p,e}^{up}$, the minimum number of active links existing required by the edge switch e of the pod p to support the up-traffic (edge to aggregation:

$F(e \to a)$):

$$LEdge_{p,e}^{up} = \left\lceil \left(\sum_{a \in A_p} F\left(e \to a\right) \right) / r \right\rceil \tag{2.2}$$

(b) In the pod p, the maximum number of links needed by each edge switch to support the up-traffic corresponds to $NAgg_p^{up}$:

$$NAgg_p^{up} = \max_{e \in E_p} \left\{ LEdge_{p,e}^{up} \right\} \tag{2.3}$$

2. $NAgg_p^{down}$ is the number of aggregation switches required to satisfy the down-traffic in the pod p:

(a) Computation of $LAgg_p^{down}$, the minimum number of links used to support down traffic from the core layer to the aggregation layer in each pod:

$$LAgg_p^{down} = \left\lceil \left(\sum_{c \in C, a \in A_p} F\left(c \to a\right) \right) / r \right\rceil \tag{2.4}$$

(b) In the pod p, every aggregation switch has $k/2$ links to the core layer, therefore the number of aggregation switches required to satisfy the down-traffic can be deduced from the number of links:

$$NAgg_p^{down} = \left\lceil LAgg_p^{down} / \left(k/2\right) \right\rceil \tag{2.5}$$

Secondly, the traffic between the core layer and the most active pod is used to compute $NCore_{C_i}$, the number of core switches in the group C_i that must be active to satisfy the traffic demands. This activity is represented by the up-traffic, or more precisely by the number of active links needed between each pod and the core layer group C_i to support the traffic aggregation to core.

$$NCore_{C_i} = \max_{p \in P} \left(LAgg_{p,C_i}^{up} \right) \tag{2.6}$$

The number of links $LAgg_{p,C_i}^{up}$ is computed using the up-traffic between each aggregation switch of the pod p and all core switches of the group c connected:

$$LAgg_{p,C_i}^{up} = \left\lceil \left(\sum_{c \in C_i, a \in A_p} F\left(c \to a\right) \right) / r \right\rceil \tag{2.7}$$

Finally, the parameter r is used to improve the robustness of the method by setting the desired link utilization. Reducing r reserves part of the resources to support traffic overloads. When $r = 1$, the above calculations aim to use 100% of the link capacity.

Once finished, the number of active switches required in the core and aggregation layer to support the traffic is known. These optimizer results can be given as inputs to the forwarding module.

2.2 Forwarding

The aim of the forwarding module is to define the set of flow rules and to send them to every switch trough the controller. It should take into account the subset of active devices and the multipath topology. For this purpose and thanks to the central SDN view, routing concepts can be used to forward in layer 2 switches (OSI Layer 2).

Existing methods like ECMP (Equal Cost Multipaths Protocol) split flows along each links available by distributing traffic over all equal-cost paths. But this distribution can impact the performances of TCP traffic which is sensitive to the change of path. Hence, the choice to have a single default path between each pair of hosts was made.

2.2.1 Network IP Addresses

To define IP addresses, the idea is the following: to create different sub-networks depending on the position of each host in the fat-tree topology. We decided to use `10.0.0.0/8` as the network address. Each pod of the network is identified through the 8 following bits of the IP address to form the sub-network : `10.p.0.0/16`. Then, the next 8 bits are used to specify the position `e` of the edge switch in the current pod (left to right order), the IP address of this sub-network is `10.p.e.0/24`. Finally, the last 8 bits are used by the position `h` (in the edge sub-network) of the host connected to the edge switch `e` to create an IP address of the form `10.p.e.h`. For example, figure 2.2 represents IP network and sub-networks in a $k = 4$ fat-tree topology. These IP address specificities will be used to define default paths.

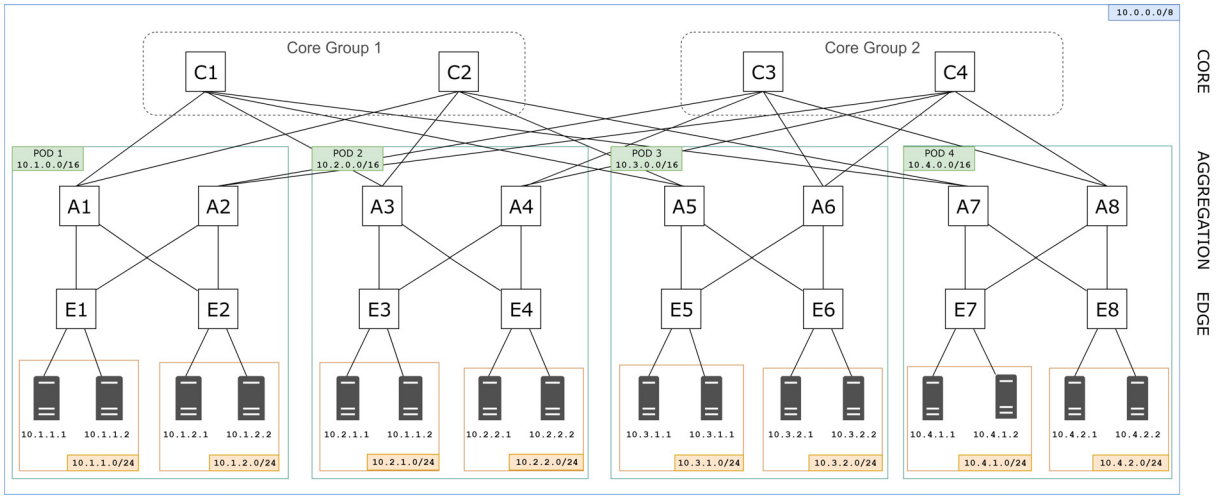

Figure 2.2: IP Networks in $k = 4$ Tat-Tree Topology

2.2.2 Default Paths

In order to have a single path between every pair of hosts and optimally use multiple paths present in the topology, default paths are pushed when the network start-up. The idea is to distribute paths between hosts to maximize the bandwidth available for host-to-host's traffic. A default path is composed of a set of flow rules to enable the connexion between two hosts.

To define them, down-traffic and up-traffic are differentiated. The down-traffic is the traffic that goes inside of the current sub-network and the upward traffic is the traffic that goes outside of the current sub-network.

To generate them, the IP destination and IP source are used respectively, by flow rules for down-traffic and for up-traffic. This technique makes it possible to distribute the traffic on every available links. In order to reduce the number of flow rules, netmasks are used. Every layer of switches matches a certain number of bits of the IP address. This number corresponds to the netmask of the following or previous sub-network, respectively for IP destination and IP source. For example, in the $k = 4$ fat-tree topology shown in the figure 2.2, core switches match flow rules with the 16 first bits of the IP address and aggregation switches match with the 24 first bits. In addition, a higher priority is given to down flow rules in order to match them before up flow rules to solve the overlap problem. For example, the figure 2.3 represents two flows crossing the switch A1. And the table 2.1 is the OpenFlow table of the switch A1 after the installation of the default path in the $k = 4$ fat-tree. The two first rules are used to reach the two connected sub-network (down-traffic). And the two last ones allow incoming traffic to be sent through the two links available up to the core layer in order to reach the rest of the network.

Selector Type	IP	Flow Priority	Output Port
IP Destination	10.1.2.0/24	4	1
IP Destination	10.1.1.0/24	4	2
IP Source	10.1.2.0/24	3	4
IP Source	10.1.1.0/24	3	3

Table 2.1: Extract OpenFlow Table Switch A1 (in $k = 4$ fat-tree)

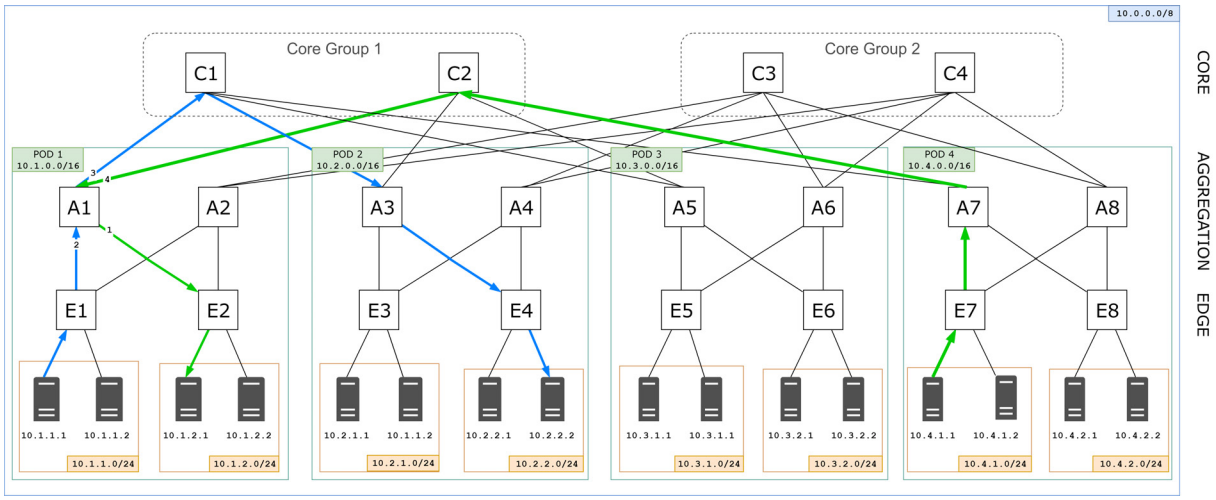

Figure 2.3: Graphic Representation of Two Flows crossing A1 (in $k = 4$ fat-tree)

2.2.3 Flow Rules Management

When traffic is running in the network, the optimizer module collects traffic statistics in order to compute the minimal number of switches needed to satisfy the demand. Then the forwarding module generates a new set of default paths based on the subset of devices received. After removing the previous default paths, the algorithm used is the same as before. The only thing that varies is the number of aggregation and core switches available.

2.3 Power control

The power control is the last module used in our application. The aim of this one is to manage the power of every switch by turning off unneeded devices and turning on needed ones. But its use is only possible with real switches. Therefore, the power control module will be developed in the *Real Test-bed* part.

3 Mininet Emulation

During the implementation of the ElasticTree algorithms, the Mininet network emulator was used to validate the correct behavior and to run some experiments. For this purpose, specific $k = 4$ and $k = 8$ fat-tree topologies were implemented and traffic was generated.

3.1 System Architecture

As seen before, the optimizer computations assume homogeneous fat-tree (for simplicity). Here, a homogeneous fat-tree is defined with a full-bisection-bandwidth topology with three layers of similar switches and $1Gbit/sec$ links. Every network device is a small virtual switch with k ports. The choice has been made to have two different topologies:

1. A **smaller configuration** with **4-port switches**, this is a degree 4 fat-tree topology. It is composed of 16 hosts and 20 switches. The figure 1.1 is an illustration of this one.
2. A **larger configuration** with **8-port switches**, this is a degree 8 fat-tree configuration. It is composed of 128 hosts and 80 switches.

The figure 3.1 represents the entire architecture of our emulated network. The data center network is generated with a Python script using the mid-level API of Mininet (because it does not propose pre-generated fat-tree topologies). Then, the external controller (ONOS) is added to enable the network control through the OpenFlow protocol. And our ElasticTree application can run independently using HTTP requests to communicate with the controller. From this application, the forwarding module is used at the starting to send default paths (between every pair of hosts)and after the execution of the optimizer, to generate new default paths based on the sub-set of devices received. It can be noted that we do not power off inactive switches due to the use of virtual switches in a single machine. Therefore, the power control module is not necessary in this setup.

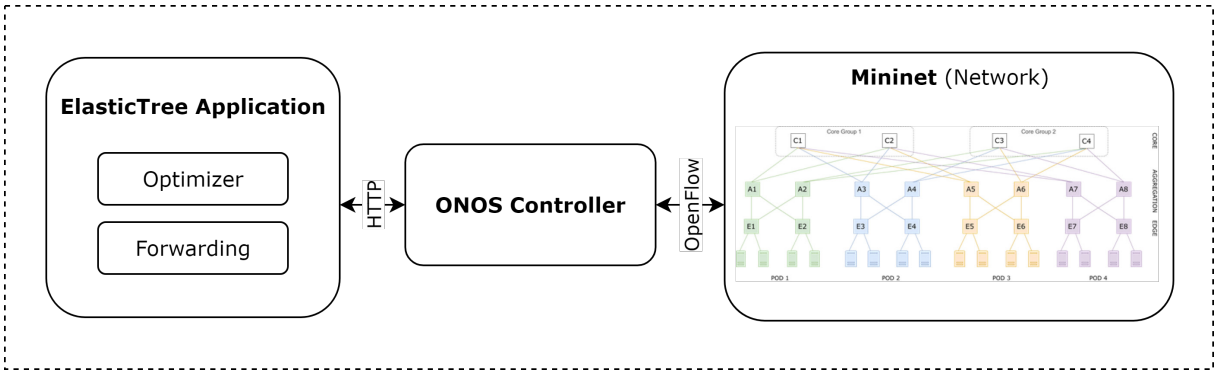

Figure 3.1: Mininet Architecture

3.2 Traffic patterns

To illustrate the performances of our ElasticTree application, we introduce three use cases. They represent possible energy savings over three communication patterns. First, two uniform traffic demands and then, a more realistic one (sine-wave demand) will be detailed. These traffic patterns were defined in the published literature [1] and are useful to have another validation of our ElasticTree implementation.

3.2.1 Uniform Demand, Varying Locality

The uniform traffic can be reached when all hosts communicate with only one other host (one flow per host), with the same bandwidth utilization in our simulation. Two extreme locality cases are considered:

- The **near traffic** which is highly localized where hosts communicate only with hosts in the same pod through their edge switch (e.g. $host1 \rightarrow host2$, $host3 \rightarrow host4 \ldots$);
- The **far traffic** which is non-localized where hosts communicate only with hosts in other pods through core switches (e.g. $host1 \rightarrow host5$, $host2 \rightarrow host6 \ldots$).

In these traffic patterns, all traffic stays within the data center network and never go outside. Their aim is to demonstrate extreme cases of energy saving. More realistic traffic will be defined in the following part (3.2.2).

3.2.2 Sine Wave Demand

The utilization of a data center varies over time, on daily, weekly, or annually time scales, for example. From this observation, the sine-wave demand was designed to simulate this specificity. The traffic generated follows the mathematical curve of a sine and the locality varies (near or far traffic). The figure 3.2 shows the variation of the demand as a function of time. The time is intentionally kept without unit because each period of time can be different according to the scale.

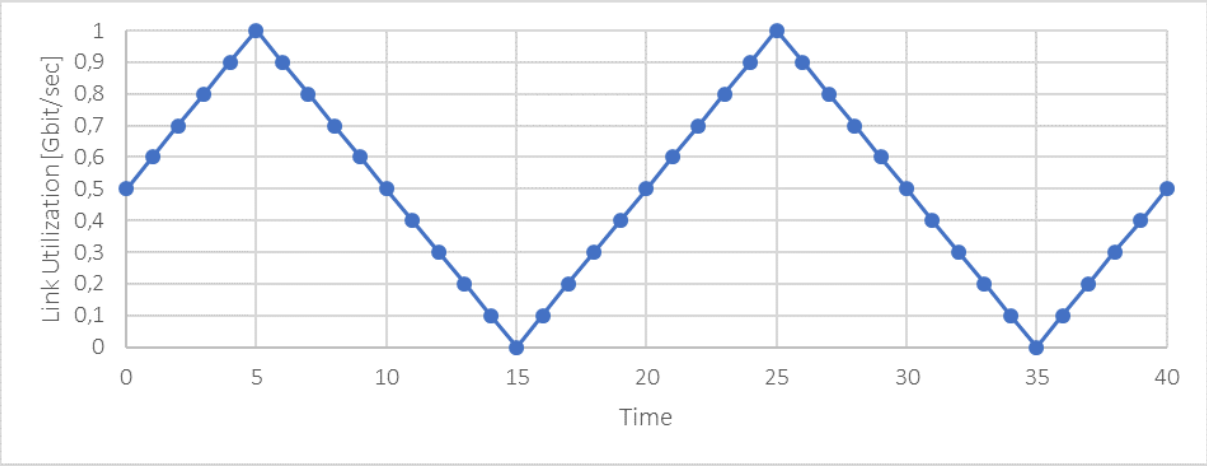

Figure 3.2: Sine-Wave Demand

3.3 Results

In order to generate these three types of traffic and have some results, the *iperf* tool in UDP mode was used. The topology utilised for these experiments is the $k = 4$ fat-tree with $1Gbit/sec$ links. The larger one ($k = 8$) could not be used because my computer resources were not good enough to support large scale experiments. The metric used to measure performances is the network utilization in percentage:

$$\text{Network utilization } (\%) = \frac{\text{Number of active switches}}{\text{Total number of switches}} \tag{3.1}$$

And the bandwidth represents the traffic sent by each host to another (each `iperf` client).

The first experiments were realized to analyze the network utilization for **uniform near and far traffic**. Figure 3.3 shows the results. The near traffic does not affect the network utilization because it is satisfied by edge layer switches. By this way, the Minimum Spanning Tree (MST) is always sufficient, which would enable the traffic to use only 65% of the network. If inactive switches are considered to be non-energy consuming, near traffic and our ElasticTree application can save 35% of the energy consumed only by the network (in a $k = 4$ fat-tree topology). In contrast, the far traffic requires the full configuration when the bandwidth is above $0,5Gbit/sec$, which is the half capacity of a link. This case of use is not in favor of energy saving. Finally, with these experiments, the benefits of local communication in a data center using the ElasticTree application are highlighted.

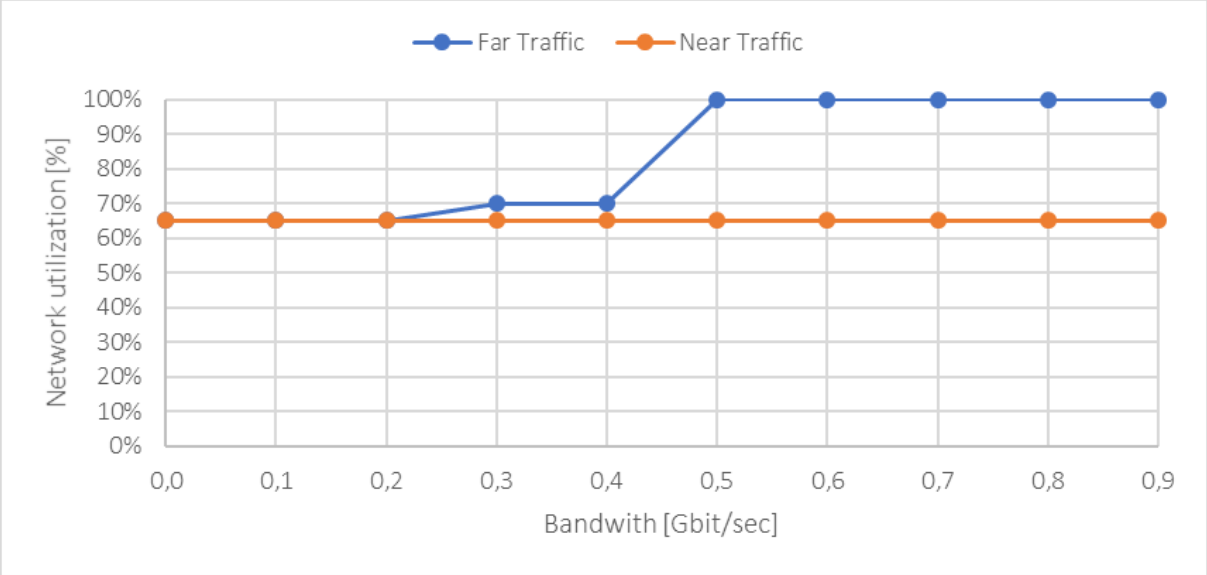

Figure 3.3: Network Utilization for Uniform Traffic Pattern in $k = 4$ Fat-Tree Emulated in Mininet

The second experiment was realized to analyze the network utilization for **sine-wave traffic** as a function of time. Based on the previous analysis, the near traffic is still a great case because it does not affect the network utilization. For this reason, more research are not valuable. In contrast, the far traffic, that alters the use of the network, is more interesting. Figure 3.4 shows results of varying far traffic demand. It can be observed that the network utilization follows the traffic demand from MST (65% of the network utilization) to the full configuration (100%). On 40 periods of time, the average demand is $0,5Gbit/sec$, and the average network

utilization is 86%. It demonstrates that even with far traffic, our ElasticTree application is able to save, on average, 14% of the energy consuming by the switch infrastructure.

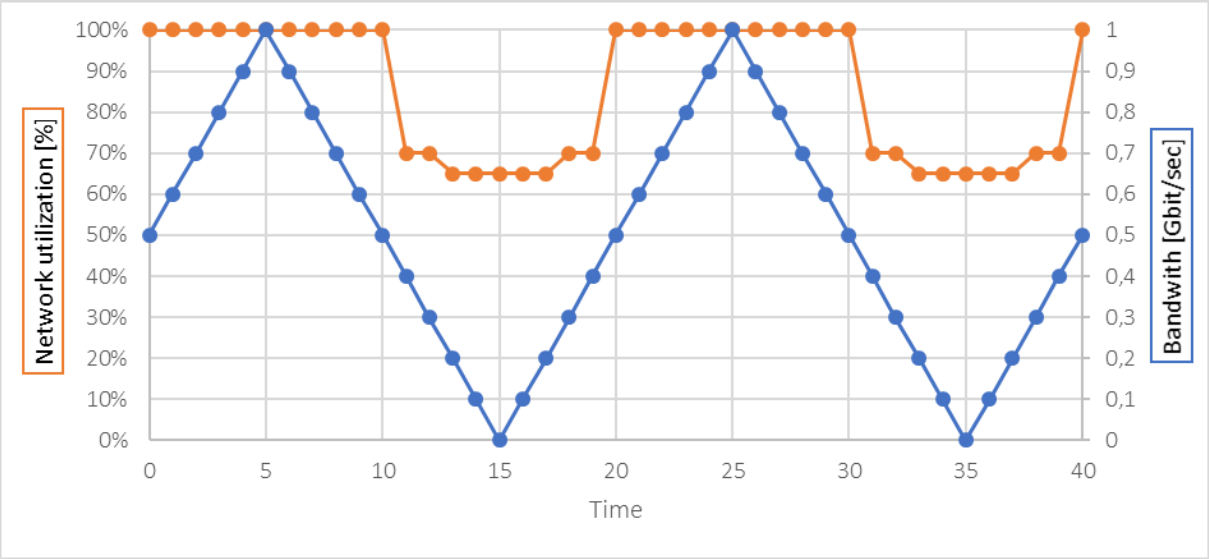

Figure 3.4: Network Utilization for Far Sine-Wave Traffic Pattern in $k = 4$ Fat-Tree Emulated in Mininet

The obtained results made it possible to evaluate and validate the implementation of the ElasticTree application on an emulated network. It can be noted that results obtained are relevant for a $k = 4$ fat-tree network and could have been improved significantly with a larger topology.

4 Real Test-bed

As seen previously, network emulators (e.g. Mininet) can be used to create virtual prototypes of SDN in a single machine. But test-beds enable the validation of applications under real conditions. Therefore, the third part of the project was to build a low-cost test-bed for SDN data center analysis. For this purpose, Raspberry-Pis were used. They are low-cost and small computers which can host virtual switches. The network topology created (figure 4.3) is half of a $k = 4$ fat-tree, with 10 switches rather than 20. This prototype is inspired by a previous project of my supervisor [2]. A video of the construction was realized and is available on *YouTube* `https://youtu.be/my_5TS-8hLI`.

In this part, the architecture of the platform is presented in two sections: first the physical infrastructure of the test-bed, and secondly the software stack. Finally, results obtained are introduced.

4.1 System Architecture

4.1.1 Physical Architecture

4.1.1.1 Tower Hardware

This test-bed of small SDN data center should be low-cost. To achieve this goal, the use of cost-efficient devices is needed. A Raspberry-Pi computer is designed to this end. In addition, it can host a virtual switch and its 4 USB ports can be used as ethernet port with adaptors. The choice of the Raspberry-Pi 3 model B+ was made for two main reasons:

- the USB 3.0 ports which enable best throughput performances, compared to USB 2.0 ports;
- the low-power consumption, compared to the model 4.

Furthermore, heatsinks are used to improve the heat dissipation and the performances. The figure 4.1 is a picture of one Raspberry-Pi and the figure 4.2 is the tower built (using 10 Raspberry-Pis).

To provide communication in the test-bed and to separate the data center traffic from the control traffic, two different networks are used:

- *Red cables*: the **data network** used for data communication in the network (server to server and client to server);
- *Green cables*: the **control network** used for internal communication between the SDN controller and switches.

The network is a $300Mbit/sec$ network with a small $k = 4$ fat-tree topology represented in the figure 4.3. Every Raspberry-Pi includes a virtual switch (Open vSwitch) using 4 USB ports with ethernet adaptors (*TP-LINK UE300*) connected to the data network. The integrated ethernet port is used to connect them to the controller (through the control network) in order

Figure 4.1: Picture of a Raspberry-Pi 3 B+

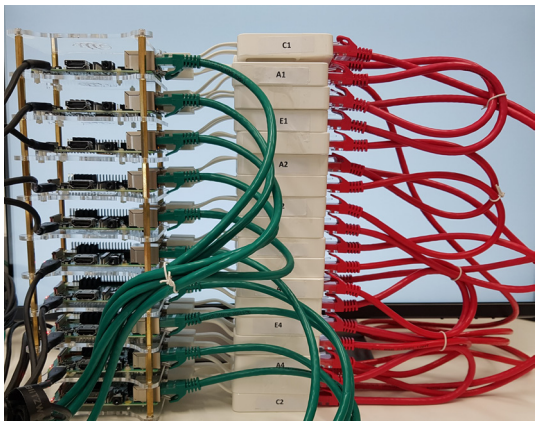

Figure 4.2: Picture of the Tower Built

to transfer OpenFlow requests. With this organization, the control network is not impacted by the amount of traffic in the data network and vice-versa.

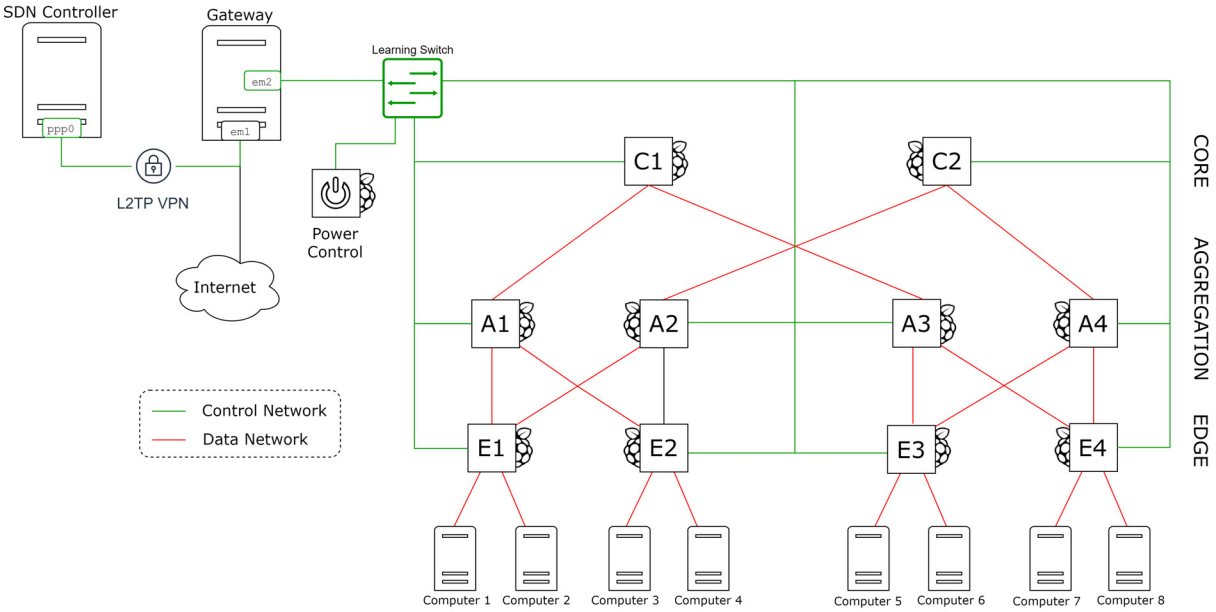

Figure 4.3: System Architecture of the Test-bed

4.1.1.2 Power Control Hardware

The first case of use of this test-bed is to carry out experiments with the ElasticTree application previously presented. For this purpose, the power control module is needed to manage the energy consumption. In contrast with a lot of devices, Raspberry-Pis are not compatible with Wake-on-LAN standard[1]. To compensate this lack, a power control using a multiple channels relay was used. Our 16 channels relay, represented in the figure 4.4, operates like an electrical switch to manage the energy given to every Raspberry-Pi. Another Raspberry-Pi controls it using the row of GPIO[2]. It enables the sending of electrical pulses to the relay to connect or

[1]Wake-on-LAN (WoL) is an Ethernet standard that allows a computer to be turned on or awakened by a network message. *Source: Wikipedia*

[2]A General-Purpose Input/Output (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board whose behavior is controllable by the user at run time. *Source: Wikipedia*

disconnect an electrical switch. The figure 4.4 shows the Raspberry-Pi used for control and the relay with 8 channels connected.

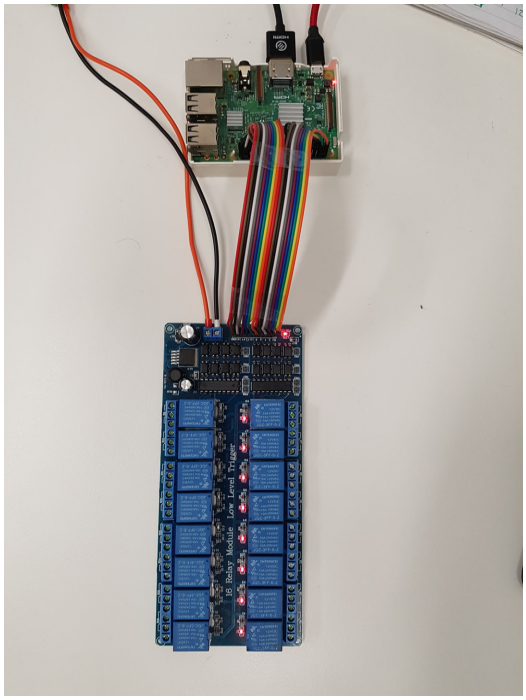

Figure 4.4: Picture of the 16 Channels Relay and the Raspberry-Pi Power Control

Figure 4.5: Picture of the Existing Infrastructure and the Network of Switches

Then, in order to measure the energy consumed by the network of switches, Raspberry-Pis are connected to two vertical managed enclosure Power Distribution Units (ePDUs). Each ePDU is used to measure the energy efficiency of the test-bed when the ElasticTree application is running.

Finally, the table 4.1 is a summary of the hardware setup of the test-bed. It may be noticed that servers and ePDUs are not include because we assume to reuse an existing infrastructure shows in the figure 4.5.

Description	**Quantity**
Raspberry Pi 3 Model B+	10
HeatSink	10
Micro SD card	10
Tower Mount Enclosure 10 levels	1
Network	
TP LINK UE300 adaptors	40
Ethernet cable	33
Learning switch 16 ports	1
Power	
Anker PowerPort 6 ports	2
Micro USB cable	10
Relay 16 channels with power supply	1

Table 4.1: Harware Setup of the Test-bed

4.1.2 Software Architecture

In order to carry out of experiments on the presented hardware, some software configurations are required. Their descriptions are divided in three parts: first, the IP networks definition will be discussed, then the Raspberry-Pi configuration, and finally, the SDN controller software stack.

As explained previously, IP addresses were required for the forwarding module of the ElasticTree application. The figure 4.6 represents the **IP addressing system** for the data network. In addition, they were also required in the control network in order to facilitate configurations (e.g. SSH connexion). The private network `192.168.0.0/24` was used for this purpose. As shown in the figure 4.7, representing the Raspberry-Pi 5, this IP was attached to the integrated ethernet port and defined on the basis of the Raspberry-Pi ID.

To create **OpenFlow switches in Raspberry-Pis**, *Raspbian*, a Debian-based operating system for Raspberry-Pi, was used and Open vSwitch (OvS version 2.10.1) was installed. They were configured to use all USB interfaces as forwarding ports and the integrated ethernet interface as OpenFlow control port (to communicate with the controller). This configuration could be achieved by the use of the OvS bridge. As example, the internal network of the Raspberry-Pi 5 is shown in figure 4.7.

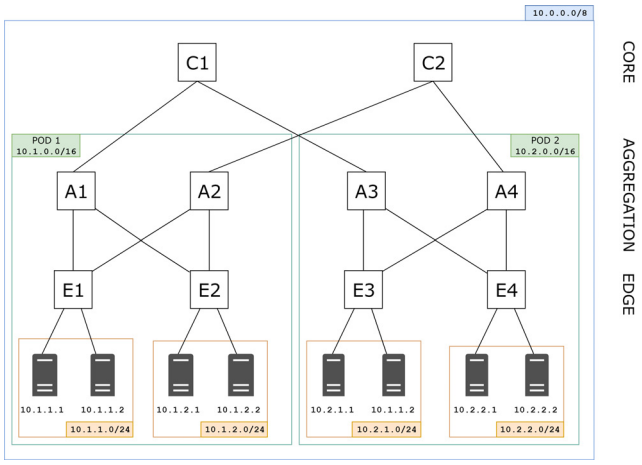

Figure 4.6: IP Networks of the Test-bed

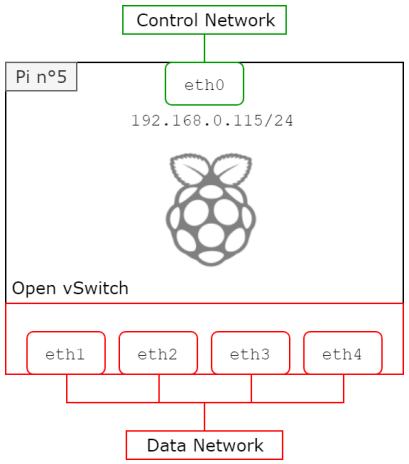

Figure 4.7: Internal Network of the Raspberry-Pi 5

Finally, the **ONOS controller** of The Linux Foundation was used (version 2.2). This software was hosted in an Ubuntu operating system (Linux). As presented in the graph of the figure 4.3, the controller could be executed remotely with the use of an existing VPN (*L2TP VPN*). The ElasticTree application was running in the same machine. As in the Mininet emulation, the optimizer and the forwarding module were present. The power control module was added to monitor the Raspberry-Pi power through the power control unit presented in part 4.1.1.2.

The last configuration performed (on one machine of the cluster) was to set up internet through the control network for configuration tacks. A **Network Address Translation** (NAT) was created, using *iptables*. In this way, a selected machine was configured as gateway for the data center.

4.2 Results

In order to evaluate the performances of the test-bed and the ElasticTree application in real environment, different use cases were used. They were introduced in the *Mininet Emulation – Traffic Patterns* part (3.2) and allow us to compare emulation and real-world experiments.

The first experiments were realized to analyze the network utilization for **uniform near and far traffic**. First of all, two types of network configuration can be distinguished:

- **100%** of the network utilization: in the **full configuration**, all switches are used to support the traffic;
- **70%** of the network utilization: in the **Minimum Spanning Tree (MST) configuration**, only the minimal number of switches is used to maintain the host to host connectivity.

Figure 4.8 shows the results. When near traffic is running in the network, MST is sufficient. The traffic is satisfied by the edge layer switches. In this case, the energy saving is maximum with 70% of network utilization. In contrast, the far traffic required quickly the full configuration. Because the current fat-tree network has only one link from the pods to the core layer, the MST configuration does not support big traffics.

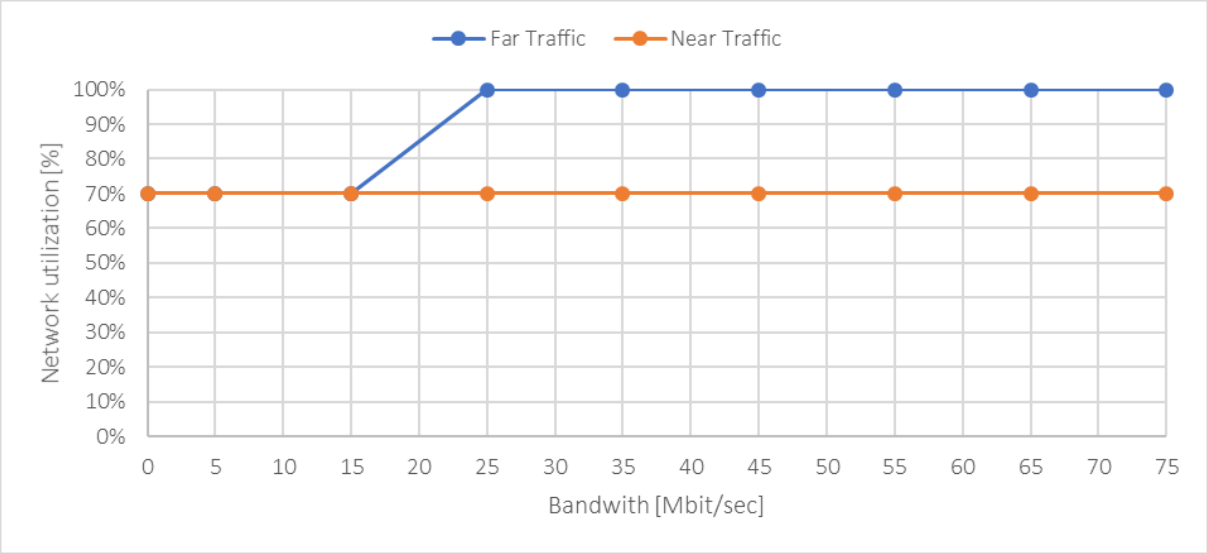

Figure 4.8: Network Utilization for Uniform Traffic in the Test-beds

The second experiment was realized to analyze the **network utilization for sine-wave traffic** as a function of time. Like in the emulation experiment, only far traffic was used, i.e. generated from each host in pod 1 to its counterpart in pod 2 ($host1 \rightarrow host5$, $host2 \rightarrow host5 \ldots$). As represented in the figure 4.9, it can be observed that the network utilization follows the traffic. Because the set of switches was limited, only two types of network configuration were used to adapt the network utilization to the traffic demand. On 40 periods of time, the average network utilization is 94%. It demonstrates that even with far traffic and a sine-wave demand, on average, **6% of energy could be saved** without deterioration of the quality of service.

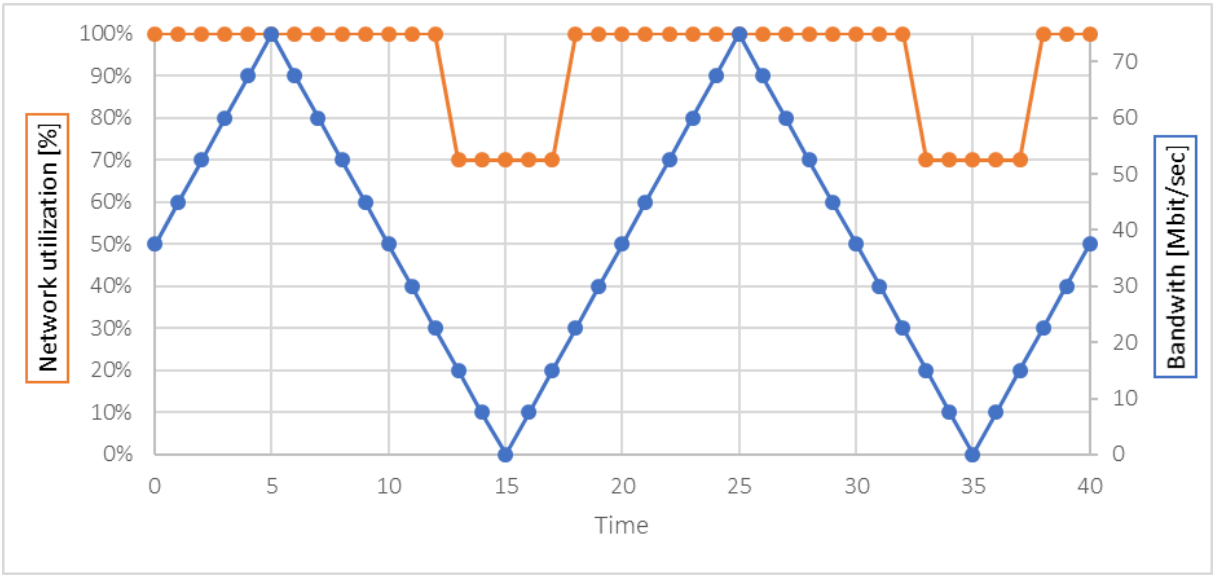

Figure 4.9: Network Utilization for Far Sine-Wave Traffic in the Test-bed

Finally, results obtained in the test-bed are homogeneous with the emulation (Mininet) ones. The behavior of the ElasticTree application implemented is validated with the latest test-bed results. But to validate the energy saving capacity of the application, **real power consumption** should be collected. For this purpose, a last experiment was realized. The idea is to analyze the total energy consumption of switches in the network. In these experiments, different amount of far uniform traffic were used. Table 4.2 details the amount of far traffic generated.

Traffic	**Amount per host [Mbit/sec]**
Low	5
Medium	15
Big	25
Huge	35

Table 4.2: Different Amount of Far Traffic Generated for Power Measures

Figure 4.10 represents the power consumed as a function of time (intentionally kept without unit). At the beginning, between $t = 0$ and $t = 17$, the network of switches was running with the full configuration (10 switches) and no traffic was generated. Then, a low traffic was produced and the power consumed increased slightly. This variation was due to the augmentation of the resources needed by each switch to forward the traffic. At $t = 29$, the ElasticTree application started and the power consumption dropped. The use of the MST configuration saved energy by switching off 3 switches. Next, the medium traffic was generated. It did not affect significantly the energy consumed because the MST was still enough. But this was no longer the case. When a big traffic was generated (from $t = 98$), the full configuration was required. It generated a power pic that corresponded to the power needed to boot-up new switches. Finally, the power consumption was similar for big and huge traffic. It could be noticed that some pics are present before each change of demand, they were generated by the manual process of traffic generation (when the *iperf* client is modified).

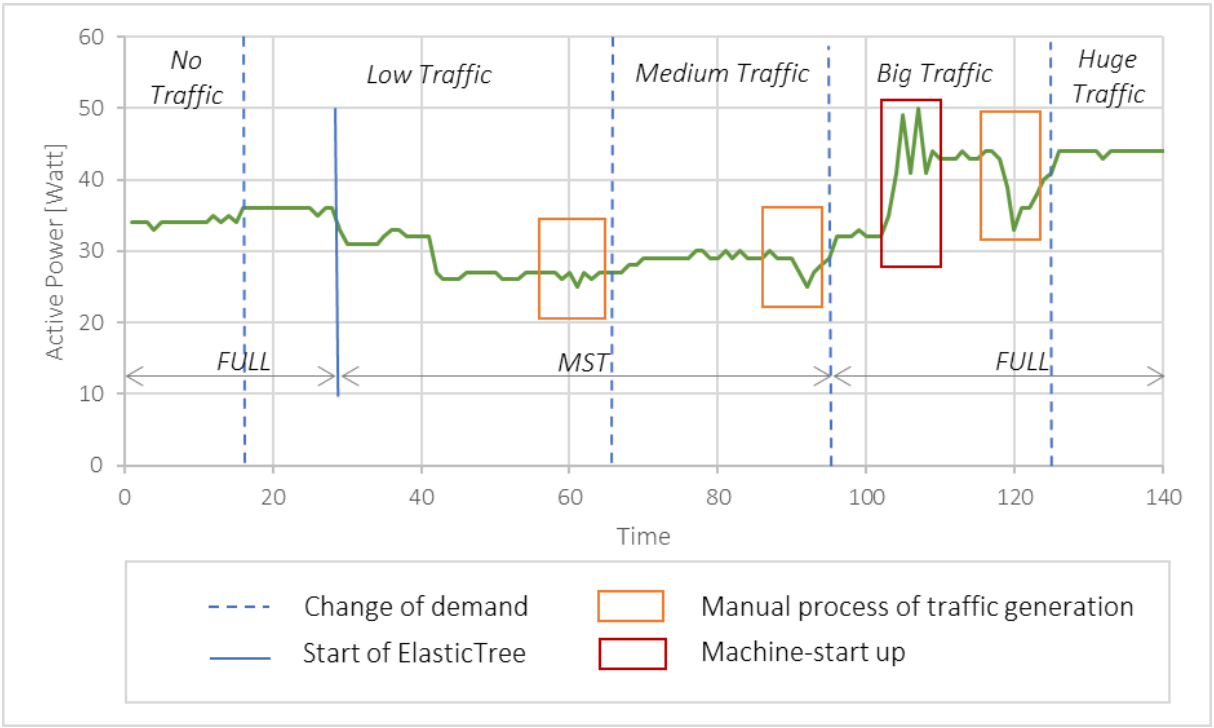

Figure 4.10: Evolution of the Power Consumed by the Raspberry-Pi Tower running OvS as a Function of Time

The obtained results in emulation and in real test-bed made it possible to fully validate the behavior of the ElasticTree application. In addition, the energy saving capacity is proven with the power consumption measures made with the test-bed. From a general point of view, energy saving results achieved are promising and augurs well for larger data center.

Conlusion

In SDN data center networks, ElasticTree is a promising solution for saving energy without compromise the quality of service. The topology patterns of a fat-tree network are useful to reduce the processing complexity of the application and to separate the forwarding module from the optimizer module. The evaluation of the application implemented was done in two steps. First, the SDN emulation software Mininet was used to validate the behavior and the scalability of the algorithms developed in a $k = 4$ and $k = 8$ fat-tree networks. Secondly, a low-cost test-bed was built using Raspberry-Pis as OpenFlow switches in a reduced-scale SDN data center. This test-bed allows us to run experiments under real conditions. Results obtained are highly positive regarding the energy consumption. Thanks to them, we were able to demonstrate the capabilities of our ElasticTree application and, at the same time, the capabilities of an SDN low-cost test-bed for energy saving applications. Based on this open source project available on *Github* (link: `https://github.com/jmathet/GreenSDN`) future developments can be done to improve the energy saving capacity of the application. A perspective could be to turn off edge switches and hosts when their resources are not required, using virtual machine migration techniques for example.

Regarding my personal feeling during this internship, it was a very rich experience. I learned a lot about the SDN network functioning, the complexity of creating low-cost test-bed and more generally, about the energy efficiency challenges in networks. In addition, it enabled me to discover the world of laboratory research. Finally, this internship allows me to explore an beautiful country during three months.

Bibliography

[1] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. 2010.

[2] A. Nadjaran Toosi, J. Son, and R. Buyya. Clouds-pi: A low-cost raspberry-pi based micro data center for software-defined cloud computing. *IEEE Cloud Computing*, 5(5):81–91, Sep 2018.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[4] Priya Mahadevan, Puneet Sharma, Sujata Banerjee, and Parthasarathy Ranganathan. A power benchmarking framework for network devices. 2009.